The Semantics of SPECTRUM[*]

Radu Grosu and Franz Regensburger

Fakultät für Informatik, Technische Universität München
80290 München, Germany

E–Mail: spectrum@informatik.tu-muenchen.de

Abstract. The SPECTRUM project concentrates on the process of developing well-structured, precise system specifications. SPECTRUM is a specification language, with a deduction calculus and a development methodology. An informal presentation of the SPECTRUM language with many examples illustrating its properties is given in [2, 3]. The purpose of this article is to describe its formal semantics.

1 Introduction

The SPECTRUM specification language is axiomatic and borrows concepts both from algebraic languages (e.g. LARCH [12]) as well as from type theoretic languages (e.g. LCF [4]). An informal presentation with many examples illustrating its properties is given in [2, 3]. We briefly summarize its principal characteristics.

Influences from algebra. In SPECTRUM specifications the influence of algebraic techniques is evident. Every specification consists of a signature and an axioms part. However, in contrast to most algebraic specification languages, the semantics of a specification in SPECTRUM is loose, i.e. it is not restricted to initial models or even term generated ones. Moreover, SPECTRUM is not restricted to equational or conditional-equational axioms, since it does not primarily aim at executable specifications. One can use full first order logic to write very abstract and non-executable specifications or only use its constructive part to write specifications which can be understood and executed as programs.

Loose semantics leaves a large degree of freedom for later implementations. It also allows the simple definition of refinement as the reduction of the class of models. This reduction is achieved by imposing new axioms which result from design decisions occurring in the stepwise development of the data structures and algorithms.

Since writing well-structured specifications is one of our main goals, a flexible language for structuring specifications has been designed for SPECTRUM. This

[*] This work is sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software" and by the German Research Community (DFG) project SPECTRUM.

structuring is achieved by using so-called specification building operators which map a list of argument specifications into a result specification. The language for these operators was originally inspired by ASL [19]. The current version borrows concepts also from Haskell [13], LARCH and PLUSS [7].

Influences from type theory. The influence from type theory is twofold. On the type level Spectrum uses shallow predicative polymorphism with type classes in the style of Isabelle [16]. The theory of type classes was introduced by Wadler and Blott [22] and originally realized in the functional programming language Haskell. Type classes may be used both to model overloading [6, 21] as well as many instances of parameterized specifications. Like in object oriented languages type classes can be organized in hierarchies such that every class inherits properties from its parent classes. This gives our language a weak object oriented flavor.

The other influence of type theory can be seen in the language of terms and their underlying semantics. Spectrum incorporates the entire notation for typed λ-terms. The definition of the semantics and the proof system was heavily influenced by LCF. Therefore Spectrum supports a notion for partial and non–strict functions as well as higher–order functions in the sense of domain theory. The models of Spectrum specifications are assumed to be certain continuous algebras. All the statements about the expressiveness of LCF due to its foundation in domain theory carry over to Spectrum.

Beside type classes there are also two features in the Spectrum logic which distinguish Spectrum from LCF. Spectrum uses three–valued logic and also allows in a restricted form the use of non–continuous functions for specification purposes. These non–continuous functions are an extension of predicates. They allow the specifier to express facts in a functional style which otherwise he would have to encode as a relation. The practical usefulness of these features has to be proved in case studies.

In conclusion, all the above features make Spectrum a very powerful general purpose specification language. It can be used successfully in data base applications, computationally intensive applications or even distributed applications since it can easily incorporate a theory for streams and stream processing functions [1].

The purpose of this article is to describe the formal semantics of the language Spectrum. This semantics incorporates in a uniform and coherent way the properties already mentioned. In comparison with other logics for higher order functions (e.g. LCF family) our main contributions are:

- a denotational semantics based on order sorted algebras for type classes (we are only aware of an operational semantics for Haskell),
- the use of non–continuous functions for specification purposes,
- the identification of predicates with (strong) boolean functions in the context of a three valued logic.

The paper is organized as follows. In section 2 we present some examples that show the use of the specification language SPECTRUM. In sections 3 and 4 we introduce the polymorphic signatures and the well–formed terms. In section 5 we describe the polymorphic algebras. Section 6 is devoted to the interpretation of terms in these algebras and to the notions of satisfaction and model. Finally we draw some conclusions in section 7.

Note that space limitations caused us to leave out the treatment of generation constraints and of constructs related to specifying in the large (e.g. signature morphisms, reducts and logical relations between algebras). A full treatment is given in [10].

2 Some Motivating Examples

Before discussing the more involved technicalities of SPECTRUM we present some motivating examples. They will help to better appreciate the design decisions made in SPECTRUM and to get more intuition about its syntax and semantics. We start by giving a polymorphic specification of lists.

```
LIST = {
    sort List α;                                   --Sort constructor List
    nil: List α;                                   --Constructors
    cons: α × List α → List α;
    first: List α → α;                             --Selectors
    rest: List α → List α;
    cons, first, rest strict;                      --Strictness & Totality
    cons total;
    List α freely generated by nil, cons;          --Generation
    axioms ∀ a : α, l : List α in
        first(nil) = ⊥;
        first(cons(a,l)) = a;
        rest(nil) = ⊥;
        rest(cons(a,l)) = l;
    endaxioms; }
```

The signature of this specification consists of the sort constructor List, the value constructors nil and cons and the selectors first and rest. The sort variable α is used both to indicate the unarity of the sort constructor and to abstract from a concrete element sort. As a consequence, all functions in the signature are defined polymorphically i.e. they can be used on all lists List τ where τ is an instance sort of α. Similarly, the use of the sort variable in the axioms part indicates their validity for all instantiations τ of this variable.

Although not explicitly in the **axioms** part, the **strict** and **total** declarations as well as the **freely generated by** declaration are actually first and respectively second order axioms. They have a significant influence on the structure of lists and their associated limits. Making cons strict we have obtained the *finite* ML lists. Had we not declared cons strict, we had obtained the Haskell lazy lists which can be infinite or finite.

Beside the explicitly declared signature, each specification also has an implicit, predefined one. This contains for example the non-continuous polymorphic strong equality function (or mapping) $= : \alpha \times \alpha$ **to** Bool. Note the use of **to** instead of $\rightarrow$ to mark this distinction.

A class is used in SPECTRUM to group together sorts which own a given set of functions which in turn satisfy a given set of axioms. For example the class EQ of all sorts owning a weak equality "predicate" == can be defined as follows:

```
Equality = {
    class EQ;
    .==. : α :: EQ ⇒ α × α → Bool;
    .==. strict total;
    axioms α :: EQ ⇒ ∀ a, b : α in
        (a == b) = (a = b);
    endaxioms; }
```

For each sort constructor (and in particular a nullary one) one can declare the domain and range classes. For example the following specification:

```
EqInstances = {
    enriches Equality + LIST + NAT;
    Nat :: EQ;
    List :: (EQ)EQ; }
```

declares that the sort Nat belongs to the class EQ. It also states that each sort in the class EQ is also mapped by List into a sort from EQ. The keywords **enriches** and + are "specifying-in-the-large" constructs. The **enriches** construct includes the signature and the axioms of the argument specification into the current definition. The + construct takes the union (on signatures and axioms) of the argument specifications.

Classes are used to restrict the range of the sort variables. For example suppose that we want to extend the specification LIST with a "predicate" .∈., testing whether an element is contained in a list. This can be done by using the above class EQ as follows:

```
LISTI = {
    enriches LIST + Equality;
    .∈. : α :: EQ ⇒ α × List α → Bool;
    .∈. strict total;
    axioms α :: EQ ⇒ ∀a, x : α, l : List α in
        ¬(a ∈ nil);
        a ∈ cons(x,l) ⇔ (a == x) ∨ a ∈ l;
    endaxioms; }
```

The use of the class EQ is vital here. On the one hand a total $.\in.$ function is not monotonic and as a consequence not a continuous function on non-flat sorts. On the other hand it is implicitly declared as a continuous function in the signature. Hence, allowing α to range over all possible sorts would make the above specification inconsistent. A similar problem occurs in ML where equality sorts are syntactically distinguished from the ones ranging over all possible sorts.

Classes can be built hierarchically. For example we can reuse the definition of the class EQ in defining a more restrictive class TOrder of total orders as follows:

```
TOrder = {
    enriches Equality;
    class TO subclass of EQ;
    .≤.: α :: TO ⇒ α × α → Bool;
    .≤. strict total;
    axioms α :: TO ⇒ ∀ a, b, c: α in
        a ≤ a;                          −−reflexivity
        a ≤ b ∧ b ≤ c ⇒ a ≤ c;          −−transitivity
        a ≤ b ∧ b ≤ a ⇒ a == b;         −−antisymmetry
        a ≤ b ∨ b ≤ a;                  −−totality
    endaxioms; }
```

Each sort in TOrder is required by the **subclass of** declaration to be also contained in the class EQ.

By allowing arbitrary class definitions we can achieve a certain degree of function overloading and of specification parameterization similar to OBJ. Moreover, since we can tune the extent of polymorphism for each function separately, we achieve a considerable degree of specification reuse. For example, we can easily extend the specification LISTI with a function min which takes the minimum of a list provided the elements are totally ordered as follows:

```
LISTM = {
    enriches LISTI + TOrder;
    min : α :: TO ⇒ List α → α;
    min strict;
    axioms α :: TO ⇒ ∀ e : α, s : List α in
        s ≠ nil ⇒ min(s) ∈ s ∧ (e ∈ s ⇒ min(s) ≤ e);
    endaxioms; }
```

This ends the section with examples about SPECTRUM. Now we present the technical details of the core language of SPECTRUM.

3 Signatures

As an abstraction from the concrete syntax a specification $S = (\Sigma, E)$ is a pair where $\Sigma = (\Omega, F, O)$ is a polymorphic signature and E is a set of Σ-formulae.

The definitions for Σ and for its components Ω, F and O are sketched in the text below. For a detailed presentation we refer to [10].

Definition 1 Sort Signature.
A sort–signature $\Omega = (K, \leq, SC)$ is an order sorted signature[2], where

- $(K, \leq)$ is a partial order on *kinds*,
- $SC = \{SC_{w,k}\}_{w \in (K \setminus \{map\})^*, k \in K}$ is an indexed set of *sort constructors* with monotonic functionalities i.e.:

$$(sc \in SC_{w,k} \cap SC_{w',k'}) \wedge (w \leq w') \Rightarrow (k \leq k')$$

A sort–signature must satisfy the following additional constraints:

- It is *regular, coregular* and *downward complete*. These properties[3] guarantee the existence of principal kinds.
- It includes the standard sort–signature (see below).
- All kinds except *map* and *cpo*, which are in the standard signature, are below *cpo* with respect to $\leq$. In other words, *cpo* is the top kind for all kinds a user may introduce.

Definition 2 The standard (predefined) sort–signature.
The standard sort–signature

$$\begin{aligned} \Omega_{standard} = (\; & \{cpo, map\}, \emptyset, \\ & \{\{\mathsf{Bool}\}_{cpo}, \\ & \;\{\rightarrow\}_{cpo\ cpo,\ cpo}, \\ & \;\{\mathbf{to}\}_{cpo\ cpo,\ map}, \\ & \;\{\times_n\}_{\underbrace{cpo \ldots cpo}_{n\ times},\ cpo} \\ & \} \\ ) \end{aligned}$$

contains two kinds and four sort constructors (actually, we have for each n a sort constructor $\times_n$):

- *cpo* represents the kind of *all complete partial orders*, *map* represents the kind of *all full function spaces*[4],

[2] Order kinded would be more precise; see [9, 8] for order sorted algebras.
[3] Regularity guarantees least kinds for every sort term. Coregularity and downward completeness guarantee unitary unification of sort terms. See [16, 20] for details.
[4] Complete partial orders are used to model continuous functions and full function spaces are used to model non-continuous functions. The latter ones are never implemented but are extremely useful for specification purposes. An alternative approach is to use only full function spaces in the semantics and to encode continuity of functions in the logic. In [18] HOLCF a higher order version of LCF is embedded into the logic HOL using the generic framework of Isabelle. In this thesis it is shown that the full function space and its subspace of continuous functions over cpo's can live together in one type frame without problems.

– Bool is the type of booleans, $\rightharpoonup$ is the constructor for lifted continuous function spaces, **to** is the constructor for full function spaces and $\times_n$ for $n \geq 2$ is the constructor for Cartesian product spaces.

The sort–signatures together with a disjoint family χ of sort variables indexed by kinds (a *sort context*) allows us to define the set of sort terms.

Definition 3 Sort Terms.
$T_\Omega(\chi)$ is the freely generated order kinded term algebra over χ.

Example 1 Some sort terms. Let $\mathsf{Set} \in SC_{cpo,\ cpo}$. Then the following terms are valid sort terms:

$$\mathsf{Set}\ \alpha \rightharpoonup \mathsf{Bool},\ \mathsf{Bool} \times \mathsf{Bool} \in T_\Omega(\{\alpha\}_{cpo})$$

The idea behind polymorphic elements is to describe families of non-polymorphic elements. In the semantics this is represented with the concept of the generalized cartesian product. For the syntax however there are several techniques to indicate this fact. E.g. in HOL ([5]) the type of a polymorphic constant in the signature is treated as a template that may be arbitrarily instantiated to build terms. This technique is also used for the concrete syntax of SPECTRUM. In the technical paper [10] we decided to make this mechanism explicit in the syntax, too, and introduced a binding operator Π for sorts and an application mechanism on the syntactic level. For a system with simple predicative polymorphism this is just a matter of taste. For a language with local polymorphic elements (ML–polymorphism) or even deep polymorphism such binding mechanisms are essential.

Definition 4 Π – Sort Terms.
$\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n.e \in T_\Omega^\Pi$ if:

- $e \in T_\Omega(\chi)$
- $\mathsf{Free}(e) \subseteq \{\alpha_1, \ldots, \alpha_n\}$
- $k_i \leq cpo$ for $k_i \in K$, $1 \leq i \leq n$

Note that the third condition rules out bound sort variables of kind *map*.

Example 2 A Π – Sort Term.

$$\Pi\alpha : cpo.\mathbf{Set}\ \alpha \rightharpoonup \mathsf{Bool} \in T_\Omega^\Pi$$

The idea of the template and its instantiation is made precise by Π–abstraction and application of such Π–sorts to non–polymorphic sorts $s \in T_\Omega(\chi)$.

In a signature every constant or mapping will have a sort without free sort variables. This motivates the following definition.

Definition 5 Closed Sort Terms.

$$T_\Omega = T_\Omega(\emptyset)$$
$$T_\Omega^{closed} = T_\Omega \cup T_\Omega^\Pi$$

Note that $T^{closed}_{\Omega,cpo}$ will contain valid sorts for constants while $T^{closed}_{\Omega,map}$ will contain valid sorts for mappings.
Now we are able to define polymorphic signatures.

Definition 6 Polymorphic Signature.
A polymorphic signature $\Sigma = (\Omega, F, O)$ is a triple where:

- $\Omega = (K, \leq, SC)$ is a sort–signature.
- $F = \{F_\mu\}_{\mu \in T^{closed}_{\Omega,cpo}}$ is an indexed set of constant symbols.
- $O = \{O_\nu\}_{\nu \in T^{closed}_{\Omega,map}}$ is an indexed set of mapping symbols.

It must include the standard signature
$\Sigma_{standard} = (\Omega_{standard}, F_{standard}, O_{standard})$ which is defined as follows:

- Predefined Constants ($F_{standard}$):
  - $\{\mathsf{true}, \mathsf{false}\} \subseteq F_{\mathsf{Bool}}$, $\{\neg\} \subseteq F_{\mathsf{Bool} \to \mathsf{Bool}}$,
    $\{\wedge, \vee, \Rightarrow\} \subseteq F_{\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}}$ are the boolean constants and connectives.
  - $\{\perp\} \subseteq F_{\Pi\alpha\, :\, cpo.\ \alpha}$ is the polymorphic bottom symbol.
  - $\{\mathsf{fix}\} \subseteq F_{\Pi\alpha\, :\, cpo.\ (\alpha \to \alpha) \to \alpha}$ is the polymorphic fixed point operator.
- Predefined mappings ($O_{standard}$):
  - $\{=, \sqsubseteq\} \subseteq O_{\Pi\alpha\, :\, cpo.\ \alpha \times \alpha\ \mathbf{to}\ \mathsf{Bool}}$ are the polymorphic equality and approximation predicates.
  - $\{\delta\} \subseteq O_{\Pi\alpha\, :\, cpo.\ \alpha\ \mathbf{to}\ \mathsf{Bool}}$ is the polymorphic definedness predicate.

4 The language of Terms

In the previous section we introduced the polymorphic signatures which serve to construct terms in the object language. The construction itself is the purpose of this section.
Like in [15] the core language used to define the semantics of Spectrum is explicitly typed i.e. the application of polymorphic constants to sort terms is explicit and the λ–bounded variables are written together with their sorts. This assures that every well formed term has a unique sort in a given context and that the semantics of this term, although given with respect to one of its derivations, is independent from the particular derivation if the sorts of the free variables are the same in all derivation contexts.

For convenience, the *concrete* language of Spectrum is like ML, HOL, LCF and Isabelle implicitly typed i.e. the type information is erased from terms. However, like all the above languages, Spectrum has principles types i.e. every implicitly typed term t has a corresponding explicitly typed term t' such that erasing all type information from t' yields again t and for every other explicitly typed term t'' having the above property, the type of t'' is an instance of the type of t' for some special notion of instance. Having principles types guaranteed, the semantics of an implicitly typed term t is simply defined to be the semantics of t'[5]. The set of well formed terms is defined in two steps. First we define the context free syntax of pre terms via a BNF like grammar. In the second step we introduce a calculus for well formed terms that uses formation rules to express the context sensitive part of the syntax.

4.1 Context Free Language (Pre Terms)

$$\begin{array}{lll}
\langle\text{term}\rangle ::= & \psi & (\textit{Variables})\\
\quad\quad | & \langle\text{id}\rangle & (\textit{Constants})\\
\quad\quad | & \langle\Pi\text{id}\rangle\ \underline{[}\{\langle\text{sortexp}\rangle\ //\underline{,}\}^{+}\underline{]} & (\textit{Polyconstant-Inst})\\
\quad\quad | & \langle\text{map}\rangle\ \langle\text{term}\rangle & (\textit{Mapping application})\\
\quad\quad | & \langle\Pi\text{map}\rangle\ \underline{[}\{\langle\text{sortexp}\rangle\ //\underline{,}\}^{+}\underline{]}\langle\text{term}\rangle & (\textit{Polymapping-Inst})\\
\quad\quad | & \underline{\langle}\{\langle\text{term}\rangle\ //\underline{,}\}^{2+}\underline{\rangle} & (\textit{Tuple}\ n \geq 2)\\
\quad\quad | & \underline{\lambda}\ \langle\text{pattern}\rangle\ \underline{.}\ \langle\text{term}\rangle & (\lambda\textit{-abstraction})\\
\quad\quad | & \langle\text{term}\rangle\ \langle\text{term}\rangle & (\textit{Application})\\
\quad\quad | & \underline{\mathbf{Q}}\ \langle\text{tid}\rangle\ \underline{.}\ \langle\text{term}\rangle & (Q \in \{\forall^{\perp}, \exists^{\perp}\})\\
\quad\quad | & \underline{(}\ \langle\text{term}\rangle\ \underline{)} & (\textit{Priority})
\end{array}$$

$$\begin{array}{llll}
\langle\text{tid}\rangle & ::= \psi\ \underline{:}\ \langle\text{sortexp}\rangle & \langle\text{sortexp}\rangle & ::= T_{\Omega}\ (\chi)\\
\langle\text{pattern}\rangle & ::= \langle\text{tid}\rangle \mid \underline{\langle}\{\langle\text{tid}\rangle\ //\underline{,}\}^{2+}\underline{\rangle} & & \\
\langle\text{id}\rangle & ::= F_{T_{\Omega,\,cpo}} & \langle\text{map}\rangle & ::= O_{T_{\Omega,\,map}}\\
\langle\Pi\text{id}\rangle & ::= F_{T^{\Pi}_{\Omega,\,cpo}} & \langle\Pi\text{map}\rangle & ::= O_{T^{\Pi}_{\Omega,\,map}}
\end{array}$$

In addition all object variables $x \in \psi$ are different from sort variables $\alpha \in \chi$ and all variables are different from identifiers in F and O.

4.2 Context Sensitive Language

With the pre terms at hand we can now define the well formed terms. We use a technique similar to [15] and give a calculus of formation rules. Since for sort variables there is only a binding mechanism in the language of sort terms but not in the language of object terms, we need no dynamic context for sort variables. The disjoint family χ of sort variables (the sort context) carries

[5] The advantage of this technique is that the problem of defining and finding (rsp. deciding) the principal type property is separated from the definition of the semantics. The drawback is the introduction of two languages namely the one with implicit typing and the one with explicit types. An alternative would be to define the semantics directly on well formed derivations for implicitly typed terms avoiding the introduction of an explicitly typed language. However, since the type system of SPECTRUM is an instance of the type system of Isabelle, we preferred to use an explicit type system and refer to [16, 17] for results about principal typings.

enough information. For the object variables, however, there are several binders and therefore we need an explicit variable context.

Definition 7 Sort Assertions.
The set of sort assertions $\rhd$ consists of tuples (χ, Γ, e, τ) where:

- χ is a sort context.
- $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is a set of sort assumptions (a variable context), such that $\tau_i \in T_{\Omega,cpo}(\chi)$ and no x_i occurs twice in the sort assumptions contained in Γ (valid context condition). This prohibits overloading of variables in one scope.
- e is the pre term to be sorted.
- $\tau \in T_{\Omega,cpo}(\chi)$ is the derived sort for e.

We define:
$(\chi, \Gamma, e, \tau) \in \rhd$ if and only if there is a finite proof tree D for this fact according to the natural deduction system below.

When we write $\Gamma \rhd_\chi e :: \tau$ in the text we actually mean that there is a proof tree (sort derivation) for $(\chi, \Gamma, e, \tau) \in \rhd$. If we want to refer to a special derivation D we write $D : \Gamma \rhd_\chi e :: \tau$. The intuitive meaning of the sort assertion (χ, Γ, e, τ) with $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is that if the variables x_1, ..., x_n have sorts τ_1, ..., τ_n then the pre term e is well formed and has sort τ.

Formation rules for well formed terms

Axioms:

$$(\text{var})\frac{}{x : \tau \rhd_\chi x :: \tau} \qquad\qquad (\text{const})\ \frac{}{\emptyset \rhd_\chi c :: \tau}\left\{ c \in F_\tau \right.$$

$$(\Pi\text{-inst})\frac{}{\emptyset \rhd_\chi f[\tau_1, \ldots, \tau_n] :: \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}\left\{ \begin{array}{l} f \in F_{\Pi \alpha_1 : k_1, \ldots, \alpha_n : k_n . \tau} \\ \alpha_i : k_i \Rightarrow \tau_i : k_i \end{array} \right.$$

Note that in the above axiom $f[\tau_1, \ldots, \tau_n]$ is part of the syntax
whereas $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is a meta notation for this presentation of the calculus. The axiom states that given a polymorphic constant $f \in F_{\Pi \alpha_1 : k_1, \ldots, \alpha_n : k_n . \tau}$ every instance of f via the sort expressions $\tau_i : k_i$ yields an explicitly typed term $f[\tau_1, \ldots, \tau_n]$ of sort $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ which is τ after simultaneous replacement of all sort variables α_i by sort expressions τ_i of appropriate kind.

Inference Rules:

$$(\text{weak})\frac{\Gamma \rhd_\chi e :: \tau}{\Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \rhd_\chi e :: \tau}$$

The 'valid context condition' in the rule (weak) prevents us from building contexts Γ with $x : \tau, x : \sigma \in \Gamma$ and $\tau \neq \sigma$

$$\text{(map-appl)}\ \frac{\Gamma \rhd_\chi e :: \tau_1}{\Gamma \rhd_\chi oe :: \tau_2}\left\{\ o \in \mathrm{O}_{\tau_1 \mathbf{to} \tau_2}\right.$$

$$(\Pi\text{map-appl})\frac{\Gamma \rhd_\chi e :: \sigma_1[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}{\Gamma \rhd_\chi o[\tau_1, \ldots, \tau_n]e :: \sigma_2[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}\ \{\star$$

where

$$\star = \begin{cases} o \in \mathrm{O}_{\Pi\alpha_1:k_1,\ldots,\alpha_n:k_n.\sigma_1 \mathbf{to} \sigma_2} \\ \alpha_i : k_i \Rightarrow \tau_i : k_i \end{cases}$$

The rules (map-appl) and (Πmap-appl) are the formation rules for application of (polymorphic) mappings to terms. They ensure that a symbol for a mapping alone is not a well formed term which means that mappings may only occur in application context.

$$\text{(tuple)}\ \frac{\Gamma \rhd_\chi e_1 :: \tau_1 \ldots \Gamma \rhd_\chi e_n :: \tau_n}{\Gamma \rhd_\chi \langle e_1, \ldots, e_n\rangle :: \tau_1 \times \ldots \times \tau_n}\left\{\ n \geq 2\right.$$

$$\text{(abstr)}\frac{\Gamma, x : \tau_1 \rhd_\chi e :: \tau_2}{\Gamma \rhd_\chi \lambda x : \tau_1.e :: \tau_1 \longrightarrow \tau_2}\ \{e \dagger x$$

$$\text{(patt-abstr)}\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \rhd_\chi e :: \tau}{\Gamma \rhd_\chi \lambda\langle x_1 : \tau_1, \ldots, x_n : \tau_n\rangle.e :: \tau_1 \times \ldots \times \tau_n \longrightarrow \tau}\begin{cases} e \dagger x_i \\ 1 \leq i \leq n \end{cases}$$

where $e \dagger x$ is a property of pre terms. A calculus for $e \dagger x$ is presented below.

$$\text{(appl)}\frac{\Gamma \rhd_\chi e_1 :: \tau_1 \longrightarrow \tau_2 \qquad \Gamma \rhd_\chi e_2 :: \tau_1}{\Gamma \rhd_\chi e_1 e_2 :: \tau_2}$$

Note that formations for (map-appl) (Πmap-appl) and (appl) use implicit but different application mechanisms. There is no problem in determining the last step in a derivation for a term e_1e_2. If e_1 is not a constant then rule (appl) must be used since there are no variables or composed terms for mappings. If on the other hand e_1 is a constant then the choice is also clear since F and O are disjoint. Of course there remains the problem of guessing the right type τ_1 for the term e_1 in rule (appl) if e_1 is a composed term. But this is another problem of type inference not concerning the distinction between mappings and functions in application context.

$$\text{(quantifier)}\ \frac{\Gamma, x : \tau \rhd_\chi e :: \mathsf{Bool}}{\Gamma \rhd_\chi Qx : \tau.e :: \mathsf{Bool}}\left\{\ Q \in \{\forall^\perp, \exists^\perp\}\right.$$

$$\text{(priority)}\frac{\Gamma \rhd_\chi e :: \tau}{\Gamma \rhd_\chi (e) :: \tau}$$

This concludes the definition of sort derivations. We now present the calculus for $e \dagger x$. The purpose of this side condition is to prohibit the building of λ-terms that do not have a continuous interpretation. Consider the term:

$$\lambda x:\mathsf{Bool}.{=}[\mathsf{Bool}]\langle x, x\rangle$$

In our semantics the interpretation of the symbol $=$ is the polymorphic identity which is by definition not monotonic. If we allowed the above expression as a well formed term its interpretation would have to be a non-monotonic function.

The property $e \dagger x$ is recursively defined on the structure of the pre term e. It's reading is 'e dagger x' and means 'e is continuous in x'. In the calculus below the set $\Phi(e)$ represents the set of free variables with respect to the binders $\forall^\perp$, $\exists^\perp$ and λ with the obvious definition.

$$(\dagger - \mathrm{var})\frac{}{x \dagger x}$$

$$(\dagger - \mathrm{notfree})\frac{x \notin \Phi(e)}{e \dagger x}$$

$$(\dagger - \mathrm{tuple})\frac{e_1 \dagger x \qquad \ldots \qquad e_n \dagger x}{\langle e_1, \ldots, e_n\rangle \dagger x}$$

$$(\dagger - \mathrm{abstr})\frac{e \dagger x \qquad e \dagger y}{\lambda y:\tau.e \dagger x}$$

$$(\dagger - \mathrm{patt\text{-}abstr})\frac{e \dagger x \qquad e \dagger x_1 \qquad \ldots \qquad e \dagger x_n}{\lambda\langle x_1:\tau_1, \ldots, x_n:\tau_n\rangle.e \dagger x}$$

$$(\dagger - \mathrm{appl})\frac{e_1 \dagger x \qquad e_2 \dagger x}{e_1 e_2 \dagger x}$$

$$(\dagger - \mathrm{quant})\ \frac{e \dagger x \qquad e \dagger y}{Qy:\tau.e \dagger x}\left\{\ Q \in \{\forall^\perp, \exists^\perp\}\right.$$

$$(\dagger - \mathrm{prio})\frac{e \dagger x}{(e) \dagger x}$$

As we will see later in section 6 the quantifiers get a three valued Kleene interpretation. If e is continuous in x and y also $\forall^\perp y:\tau.e$ and $\exists^\perp y:\tau.e$ are continuous in x. Therefore we can allow terms like $\lambda x:\sigma.\forall^\perp y:\tau.e$ provided the dagger test $\forall^\perp y:\tau.e \dagger x$ succeeds. For example the test $\exists^\perp y:\tau.\delta y \wedge e \dagger x$ will fail since $\delta y \wedge e \dagger y$ fails.

In the report [2, 3] we used the phrase 'where x is not free on a mappings argument position' as a context condition for the formation rules (abstr) and (patt-abstr). Looking at the example $\lambda x:\sigma.\exists^\perp y:\tau.\delta y \wedge e$ we see that this is too weak for terms with quantifiers inside.

4.3 Well formed Terms and Sentences

With the context sensitive syntax of the previous paragraph we are now able to define the notion of well formed terms over a polymorphic signature. Since we use an explicitly typed system, a well formed term is a pre term e together with a sort context χ, a variable context Γ and a sort τ.

Definition 8 Well formed terms.
Let Σ be a polymorphic signature. The set of well formed terms over Σ in sort context χ and variable context Γ with sort τ is defined as follows:

$$T_{\Sigma,\tau}(\chi,\Gamma) = \{(\chi,\Gamma,e,\tau) \mid \Gamma \rhd_\chi e :: \tau\}$$

The set of all well formed terms in context (χ,Γ) is defined to be the family

$$T_\Sigma(\chi,\Gamma) = \{T_{\Sigma,\tau}(\chi,\Gamma)\}_{\tau \in T_\Omega(\chi)}$$

In addition we define the following abbreviations:

$$T_\Sigma(\chi) = T_\Sigma(\chi,\emptyset) \quad \text{(closed object terms)}$$
$$T_\Sigma = T_\Sigma(\emptyset) \quad \text{(non-polymorphic closed object terms)}$$

Considering a well formed term $(\chi,\Gamma,e,\tau) \in T_{\Sigma,\tau}(\chi,\Gamma)$ we see that all the sort derivations $D : \Gamma \rhd_\chi e :: \tau$ for this term can only differ in the applications of the formation rule (weak). Due to the vast type information contained in our pre terms e there are no other possibilities for different sort derivations.

In section 6 we will define the interpretation of a well formed term (χ,Γ,e,τ) in set $T_{\Sigma,\tau}(\chi,\Gamma)$ with respect to the inductive structure of some sort derivation for this term. To guarantee the uniqueness of our definition we now distinguish the unique and always existing normal form of a sort derivation.

Definition 9 Normal Sort Derivation.
Let $(\chi,\Gamma,e,\tau) \in T_{\Sigma,\tau}(\chi,\Gamma)$ be a well formed term. The Normal Sort Derivation $ND : \Gamma \rhd_\chi e :: \tau$ is that derivation where introductions of sort assumptions via the formation rule (weak) occur as late as possible.

A formal definition of the normal form together with a proof for the existence and uniqueness result is pretty obvious. A thorough discussion of a slightly different technique containing all the definitions and proofs can be found in [14]. Next we define formulae $\mathbf{Form}(\Sigma,\chi,\Gamma)$ and sentences $\mathbf{Sen}(\Sigma,\chi)$ over a polymorphic signature Σ and sort context χ. In SPECTRUM the set of formulae $\mathbf{Form}(\Sigma,\chi,\Gamma)$ is the set of well formed terms in context (χ,Γ) of sort Bool. This leads to a three valued logic. The sentences are as usual the closed formulae.

Definition 10 Formulae and Sentences.

$$\mathbf{Form}(\Sigma,\chi,\Gamma) = T_{\Sigma,\mathsf{Bool}}(\chi,\Gamma)$$
$$\mathbf{Sen}(\Sigma,\chi) = \mathbf{Form}(\Sigma,\chi,\emptyset) \quad \text{(closed formulae are sentences)}$$
$$\mathbf{Sen}(\Sigma) = \mathbf{Sen}(\Sigma,\emptyset) \quad \text{(non-polymorphic sentences)}$$

Example 3 Some formulae.

$$\forall^{\perp} x : \alpha . =[\alpha]\langle x, x\rangle \in \mathbf{Sen}(\Sigma, \{\alpha\})$$

$$\forall^{\perp} x : \mathbf{Nat} . =[\mathbf{Nat}]\langle x, x\rangle \in \mathbf{Sen}(\Sigma)$$

Definition 11 Specifications.
A polymorphic specification $S = (\Sigma, E)$ is a pair where $\Sigma = (\Omega, F, O)$ is a polymorphic signature and $E \subseteq \mathbf{Sen}(\Sigma, \chi)$ is a set of sentences for some sort context χ.

5 Algebras

The following definitions are standard definitions of domain theory (see [11]). We include them here to get a self-contained presentation.

Definition 12 Partial Order.
A partial order $\mathcal{A}$ is a pair $(A, \leq)$ where A is a set and $(\leq) \subseteq A \times A$ is a reflexive, transitive and antisymmetric relation.

Definition 13 Chain Complete Partial Order.
A partial order $\mathcal{A}$ is ω-chain complete iff every chain $a_1 \leq \ldots \leq a_n \leq \ldots$, $n \in \mathbb{N}$ has a least upper bound in $\mathcal{A}$. We denote it by $\sqcup_{i \in \mathbb{N}} x_i$.

Definition 14 Pointed Chain Complete Partial Order (PCPO).
A chain complete partial order $\mathcal{A}$ is pointed iff it has a least element. In the sequel we denote this least element by uu_A.

Definition 15 Monotonic Functions.
Let $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ be two PCPOs. A function[6] $f \in B^A$ is *monotonic* iff

$$d \leq_A d' \;\Rightarrow\; f(d) \leq_B f(d')$$

Definition 16 Continuous Functions.
A monotonic function f between PCPOs $\mathcal{A}$ and $\mathcal{B}$ is continuous iff for every ω-chain $a_1 \leq \ldots \leq a_n \leq \ldots$ in $\mathcal{A}$:

$$f(\bigsqcup_{i \in \mathbb{N}} a_i) = \bigsqcup_{i \in \mathbb{N}} f(a_i)$$

Since f is monotonic and $\mathcal{A}$ and $\mathcal{B}$ are PCPOs the least upper bound on the right hand side exists.

Definition 17 Product PCPO.
If $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ are two PCPOs then the product PCPO $\mathcal{A} \times \mathcal{B} = (A \times B, \leq_{A \times B})$ is defined as follows:

[6] We write B^A for all functions from A to B.

- $A \times B$ is the usual cartesian product of sets,
- $(d, e) \leq_{A\times B} (d', e') \quad iff \quad (d \leq_A d) \wedge (e \leq_B e')$,
- $uu_{A\times B} = (uu_A, uu_B)$

This definitions may be generalized to n-ary products in a straight forward way.

Definition 18 Function PCPO.
If $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ are two PCPOs then the function PCPO $\mathcal{A}\overset{c}{\rightarrow}\mathcal{B} = (A\overset{c}{\rightarrow}B, \leq_{A\overset{c}{\rightarrow}B})$ is defined as follows:

- $A\overset{c}{\rightarrow}B$ is the set of all continuous functions from A to B,
- $f \leq_{A\overset{c}{\rightarrow}B} g \quad iff \quad \forall a \in A.f(a) \leq_B g(a)$,
- $uu_{A\overset{c}{\rightarrow}B} = \lambda x : A.uu_B$

Definition 19 Lift PCPO.
If $\mathcal{A} = (A, \leq_A)$ is a PCPO then the lifted PCPO $\mathcal{A}\,\mathbf{lift} = (A\,\mathbf{lift}, \leq_{A\,\mathbf{lift}})$ is defined as follows:

- $A\,\mathbf{lift} = (A \times \{0\}) \cup \{uu_{A\,\mathbf{lift}}\}$ where $uu_{A\,\mathbf{lift}}$ is a new element which is not a pair.
- $(x, 0) \leq_{A\,\mathbf{lift}} (y, 0) \quad iff \quad x \leq_A y$

  $\forall z \in A\,\mathbf{lift}.uu_{A\,\mathbf{lift}} \leq_{A\,\mathbf{lift}} z$
- We also define an extraction function $\downarrow$ from $A\,\mathbf{lift}$ to A such that

$$\downarrow uu_{A\,\mathbf{lift}} = uu_A \qquad ; \qquad \downarrow (x, 0) = x$$

We will call the PCPOs also domains (note that in the literature domains are usually algebraic directed complete po's [11]).

5.1 The Sort Algebras

Definition 20 Sort–Algebras.
Let $\Omega = (K, \leq, SC)$ be a sort–signature. An Ω–algebra $\mathcal{SA}$ is an order sorted algebra[7] of domains i.e.:

- For the kind $cpo \in K$ we have a set of domains $cpo^{\mathcal{SA}}$. For the kind $map \in K$ we have a set of full functions spaces $map^{\mathcal{SA}}$.
- For all kinds $k \in K$ with $k \leq cpo$ we have a nonempty subset $k^{\mathcal{SA}} \subseteq cpo^{\mathcal{SA}}$.
- For all kinds $k_1, k_2 \in K$ with $k_1 \leq k_2$ we have $k_1^{\mathcal{SA}} \subseteq k_2^{\mathcal{SA}}$.
- For each sort constructor $sc \in SC_{k_1\ldots k_n,k}$ there is a domain constructor $sc^{\mathcal{SA}} : k_1^{\mathcal{SA}} \times \ldots \times k_n^{\mathcal{SA}} \rightarrow k^{\mathcal{SA}}$ such that if $sc \in SC_{w,s} \cap SC_{w',s'}$ and $w \leq w'$ then
$$sc^{\mathcal{SA}}_{w',s'} \mid_{w^{\mathcal{SA}}} = sc^{\mathcal{SA}}_{w,s}$$
  In other words overloaded domain constructors must be equal on the smaller domain $w^{\mathcal{SA}} = k_1^{\mathcal{SA}} \times \ldots \times k_n^{\mathcal{SA}}$ where $w = k_1 \ldots k_n$.

[7] See [9, 8].

We further require the following interpretation for the sort constructors occurring in the standard sort–signature:

- $\mathsf{Bool}^{\mathcal{SA}} = (\{uu_{\mathsf{Bool}}, \mathit{ff}, \mathit{tt}\}, \leq_{\mathsf{Bool}})$ is the flat three–valued boolean domain.
- For $\times_n^{\mathcal{SA}} \in cpo^{\mathcal{SA}} \times \ldots \times cpo^{\mathcal{SA}} \rightarrow cpo^{\mathcal{SA}}$:

  $$\times_n^{\mathcal{SA}}(d_1, \ldots, d_n) = d_1 \times \ldots \times d_n,\ n \geq 2$$

  is the n-ary cartesian product of domains.
- For $\rightarrow^{\mathcal{SA}} \in cpo^{\mathcal{SA}} \times cpo^{\mathcal{SA}} \rightarrow cpo^{\mathcal{SA}}$:

  $$\rightarrow^{\mathcal{SA}}(d_1, d_2) = (d_1 \overset{c}{\rightarrow} d_2)\mathbf{lift}$$

  is the lifted domain of continuous functions. We lift this domain because we want to distinguish between $\perp$ and $\lambda x.\ \perp$.
- For $\mathbf{to}^{\mathcal{SA}} \in cpo^{\mathcal{SA}} \times cpo^{\mathcal{SA}} \rightarrow map^{\mathcal{SA}}$

  $$\mathbf{to}^{\mathcal{SA}}(d_1, d_2) = d_2^{d_1}$$

  is the full space of functions between d_1 and d_2.

Definition 21 Interpretation of sort terms.
Let $\nu : \chi \rightarrow \mathcal{SA}$ be a sort environment and $\nu^* : T_{\Omega}(\chi) \rightarrow \mathcal{SA}$ its homomorphic extension. Then $\mathcal{SA}[.]\nu$ is defined as follows:

- $\mathcal{SA}[e]\nu = \nu^*(e)$ **if** $e \in T_{\Omega}(\chi)$
- $\mathcal{SA}[\Pi \alpha_1 : k_1, \ldots, \alpha_n : k_n.e] =$
  $\{f \mid f(\nu(\alpha_1), \ldots, \nu(\alpha_n)) \in \mathcal{SA}[e]\nu$ **for all** $\nu\}$

For closed terms we write for $\mathcal{SA}[e]$ also $e^{\mathcal{SA}}$.

Sort terms in T_{Ω}^{Π} are interpreted as generalized cartesian products (dependent products). By using *n-ary* dependent products we can interpret Π–terms in one step. This leads to simpler models as the ones for the polymorphic λ–calculus.

5.2 Polymorphic Algebras

Definition 22 Polymorphic Algebra.
Let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ the sort–signature. A polymorphic Σ–algebra $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ is a triple where:

- $\mathcal{SA}$ is an Ω sort algebra,
- $\mathcal{F} = \{\mathcal{F}_\mu\}_{\mu \in T_{\Omega, cpo}^{closed}}$ is an indexed set of constants (or functions), with:

  $$\mathcal{F}_\mu = \{f^{\mathcal{A}} \in \mu^{\mathcal{SA}} \mid f \in F_\mu\}$$

  such that if $f \in F_\mu$ is not the constant $\perp \in \Pi\alpha : cpo.\alpha$ then its interpretation $f^{\mathcal{A}}$ is different from uu in $\mu^{\mathcal{SA}}$. If f is polymorphic then all its instances must be different from the corresponding least element.

– $\mathcal{O} = \{\mathcal{O}_\nu\}_{\nu \in T^{closed}_{\Omega,map}}$ is an indexed set of mappings, with:

$$\mathcal{O}_\nu = \{o^{\mathcal{A}} \in \nu^{\mathcal{SA}} \ \mid \ o \in O_\nu\}$$

We further require a fixed interpretation for the symbols in the standard signature. In order to simplify notation we will write $f^{\mathcal{A}}_{d_1,\ldots,d_n}$ for the instance $f^{\mathcal{A}}(d_1,\ldots,d_n)$ of a polymorphic function and $o^{\mathcal{A}}_{d_1,\ldots,d_n}$ for the instance of a polymorphic mapping $o^{\mathcal{A}}(d_1,\ldots,d_n)$.

– Predefined Mappings ($O_{standard}$):
  - $\{=,\sqsubseteq\} \subseteq O_{\Pi\alpha:cpo.\ \alpha\times\alpha\ \mathsf{to}\ \mathsf{Bool}}$ are interpreted as *identity* and *partial order*. More formally, for every domain $d \in cpo^{\mathcal{A}}$ and $x, y \in d$:

$$x \ =^{\mathcal{A}}_d \ y := \begin{cases} \mathit{tt} \text{ if } x \text{ is identical to } y \\ \mathit{ff} \text{ otherwise} \end{cases}$$

$$x \sqsubseteq^{\mathcal{A}}_d y := \begin{cases} \mathit{tt} \text{ if } x \leq_d y \\ \mathit{ff} \text{ otherwise} \end{cases}$$

  - $\{\delta\} \subseteq O_{\Pi\alpha:cpo.\ \alpha\ \mathsf{to}\ \mathsf{Bool}}$ is the polymorphic definedness predicate. For every $d \in cpo^{\mathcal{A}}$ and $x \in d$:

$$\delta^{\mathcal{A}}_d(x) := \begin{cases} \mathit{tt} \text{ if } x \text{ is different from } \mathit{uu}_d \\ \mathit{ff} \text{ otherwise} \end{cases}$$

– Predefined Constants ($F_{standard}$):
  - $\{\mathsf{true},\mathsf{false}\} \subseteq F_{\mathsf{Bool}}$ are interpreted in the $\mathsf{Bool}^{\mathcal{SA}}$ domain as follows:

$$\mathsf{true}^{\mathcal{A}} = \mathit{tt} \qquad ; \qquad \mathsf{false}^{\mathcal{A}} = \mathit{ff}$$

  - The interpretations of $\{\neg\} \subseteq F_{\mathsf{Bool}\rightarrow\mathsf{Bool}}$, $\{\wedge,\vee,\Rightarrow\} \subseteq F_{\mathsf{Bool}\times\mathsf{Bool}\rightarrow\mathsf{Bool}}$ are pairs in the lifted function spaces such that the function components behave like three–valued Kleene connectives on $\mathsf{Bool}^{\mathcal{SA}}$ as follows:

x	y	$(\downarrow \neg^{\mathcal{A}})(x)$	$x(\downarrow \wedge^{\mathcal{A}})y$	$x(\downarrow \vee^{\mathcal{A}})y$	$x(\downarrow \Rightarrow^{\mathcal{A}})y$
tt	tt	ff	tt	tt	tt
tt	ff	ff	ff	tt	ff
ff	tt	tt	ff	tt	tt
ff	ff	tt	ff	ff	tt
uu	tt	uu	uu	tt	tt
uu	ff	uu	ff	uu	uu
uu	uu	uu	uu	uu	uu
tt	uu	ff	uu	tt	uu
ff	uu	tt	ff	uu	tt

  - $\{\bot\} \subseteq F_{\Pi\alpha:cpo.\ \alpha}$ is interpreted in each domain as the least element of this domain. For every $d \in cpo^{\mathcal{SA}}$:

$$\bot^{\mathcal{A}}_d := \mathit{uu}_d$$

- $\{\mathsf{fix}\} \subseteq F_{\Pi\alpha:cpo.\,(\alpha\to\alpha)\to\alpha}$ is interpreted for each domain d as a pair $\mathsf{fix}^{\mathcal{A}}_d \in (d \to^{\mathcal{SA}} d) \to^{\mathcal{SA}} d$ such that the function component behaves as follows:

$$(\downarrow \mathsf{fix}^{\mathcal{A}}_d)(f) := \bigsqcup_{i\in\mathbb{N}} f^n(uu_d)$$

where:

$$f^0(uu_d) := uu_d$$

$$f^{n+1}(uu_d) := (\downarrow f)(f^n(uu_d))$$

Note that $\downarrow uu_{(d\overset{c}{\to}d)\,\mathbf{lift}} = uu_{(d\overset{c}{\to}d)}$ and therefore the above definition is sound.

6 Models

6.1 Interpretation of sort assertions

In this section we define the interpretation of well-formed terms. The interpretation of $(\chi, \Gamma, e, \tau) \in T_{\Sigma,\tau}(\chi, \Gamma)$ is defined inductively on the structure of the normal sort derivation $ND : \Gamma \rhd_\chi e :: \tau$. The technique used is again due to [15].

Definition 23 Satisfaction of a variable context.
Let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ and let $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ be a a polymorphic Σ–algebra.

If Γ is a variable context and

$$\begin{array}{ll} \nu = \{\nu_k : \chi_k \to k^{\mathcal{SA}}\}_{k\in K\setminus\{map\}} & \text{sort environment (order-sorted)} \\ \eta : \psi \to \bigcup\limits_{d\in cpo^{\mathcal{SA}}} d & \text{object environment (unsorted)} \end{array}$$

then η satisfies Γ in sort environment ν (in symbols $\eta \models_\nu \Gamma$) iff

$$\eta \models_\nu \Gamma \Leftrightarrow \text{for all } x:\tau \in \Gamma.\eta(x) \in \nu^\star(\tau)$$

Definition 24 Update of object environments.

$$\eta[a/x](y) := \begin{cases} a & \text{if } x = y \\ \eta(y) & \text{otherwise} \end{cases}$$

Now we define an order sorted meaning function $\mathcal{A}[\![\cdot]\!]_{\nu,\eta}$ that maps normal sort derivations $ND : \Gamma \rhd_\chi e :: \tau$ to elements in $\mathcal{A}$. Since normal sort derivations always exist and are unique this leads to a total meaning function $\mathcal{A}[\![\cdot]\!]_{\nu,\eta} : T_\Sigma(\chi, \Gamma) \to \mathcal{A}$.

Definition 25 Meaning of a sort derivation.
The meaning of a normal sort derivation $ND : \Gamma \rhd_\chi e :: \tau$ in a polymorphic algebra $\mathcal{A}$ in sort context ν and variable context Γ such that $\eta \models_\nu \Gamma$ is given by $\mathcal{A}[ND : \Gamma \rhd_\chi e :: \tau]_{\nu,\eta}$ which is recursively defined on the structure of ND. The defining clauses are given below.

Base cases:

(var) $\mathcal{A}[x:\tau \rhd_\chi x :: \tau]_{\nu,\eta} = \eta(x)$ (const) $\mathcal{A}[\emptyset \rhd_\chi c :: \tau]_{\nu,\eta} = c^{\mathcal{A}}$

(Π-inst)

$$\mathcal{A}[\emptyset \rhd_\chi f[\tau_1, \ldots, \tau_n] :: \tau]_{\nu,\eta} = \\ f^{\mathcal{A}}(\nu^\star(\tau_1), \ldots, \nu^\star(\tau_n))$$

Inductive cases:
(weak)

$$\mathcal{A}[\Gamma \cup \{x_1:\tau_1, \ldots, x_n:\tau_n\} \rhd_\chi e :: \tau]_{\nu,\eta} = \mathcal{A}[\Gamma \rhd_\chi e :: \tau]_{\nu,\eta}$$

(map-appl)

$$\mathcal{A}[\Gamma \rhd_\chi oe :: \tau_2]_{\nu,\eta} = o^{\mathcal{A}}(\mathcal{A}[\Gamma \rhd_\chi e :: \tau_1]_{\nu,\eta})$$

(Πmap-appl)

$$\mathcal{A}[\Gamma \rhd_\chi o[\tau_1, \ldots, \tau_n]e :: \sigma_2]_{\nu,\eta} = \\ o^{\mathcal{A}}(\nu^\star(\tau_1), \ldots, \nu^\star(\tau_n))(\mathcal{A}[\Gamma \rhd_\chi e :: \sigma_1]_{\nu,\eta})$$

(tuple)

$$\mathcal{A}[\Gamma \rhd_\chi \langle e_1, \ldots, e_n\rangle :: \tau_1 \times \ldots \times \tau_n]_{\nu,\eta} = \\ (\mathcal{A}[\Gamma \rhd_\chi e_1 :: \tau_1]_{\nu,\eta}, \ldots, \mathcal{A}[\Gamma \rhd_\chi e_n :: \tau_n]_{\nu,\eta})$$

(abstr)[8]

$$\mathcal{A}[\Gamma \rhd_\chi \lambda x:\tau_1.e :: \tau_1 \rightarrow \tau_2]_{\nu,\eta} = \\ \text{the } \underline{\text{unique}} \text{ pair } (f, 0) \in \nu^\star(\tau_1 \rightarrow \tau_2) \text{ with} \\ \forall a \in \nu^\star(\tau_1).f(a) = \mathcal{A}[\Gamma, x:\tau_1 \rhd_\chi e :: \tau_2]_{\nu,\eta[a/x]}$$

(patt-abstr)

$$\mathcal{A}[\Gamma \rhd_\chi \lambda\langle x_1:\tau_1, \ldots, x_n:\tau_n\rangle.e :: \tau_1 \times \ldots \times \tau_n \rightarrow \tau]_{\nu,\eta} = \\ \text{the } \underline{\text{unique}} \text{ pair } (f, 0) \in \nu^\star(\tau_1 \times \ldots \times \tau_n \rightarrow \tau) \text{ with} \\ \forall a_1 \in \nu^\star(\tau_1), \ldots, a_n \in \nu^\star(\tau_n).f((a_1, \ldots, a_n)) = \\ \mathcal{A}[\Gamma, x_1:\tau_1, \ldots, x_n:\tau_n \rhd_\chi e :: \tau]_{\nu,\eta[a_1/x_1,\ldots,a_n/x_n]}$$

[8] the †-test ensures that the clauses for (abstr) and (patt-abstr) are well defined.

(appl)

$$\mathcal{A}[\Gamma \rhd_\chi e_1 e_2 :: \tau_2]_{\nu,\eta} =\downarrow (\mathcal{A}[\Gamma \rhd_\chi e_1 :: \tau_1 \rightarrow \tau_2]_{\nu,\eta})(\mathcal{A}[\Gamma \rhd_\chi e_2 :: \tau_1]_{\nu,\eta})$$

(universal quantifier)

$$\mathcal{A}[\Gamma \rhd_\chi \forall^\perp x : \tau . e :: \mathsf{Bool}]_{\nu,\eta} =$$

$$= \begin{cases} \mathit{tt} & \text{if } \forall a \in \nu^\star(\tau).(\mathcal{A}[\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]_{\nu,\eta[a/x]} = \mathit{tt}) \\ \mathit{ff} & \text{if } \exists a \in \nu^\star(\tau).(\mathcal{A}[\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]_{\nu,\eta[a/x]} = \mathit{ff}) \\ \mathit{uu} & \text{otherwise} \end{cases}$$

(existential quantifier)

$$\mathcal{A}[\Gamma \rhd_\chi \exists^\perp x : \tau . e :: \mathsf{Bool}]_{\nu,\eta} =$$

$$= \begin{cases} \mathit{tt} & \text{if } \exists a \in \nu^\star(\tau).(\mathcal{A}[\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]_{\nu,\eta[a/x]} = \mathit{tt}) \\ \mathit{ff} & \text{if } \forall a \in \nu^\star(\tau).(\mathcal{A}[\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]_{\nu,\eta[a/x]} = \mathit{ff}) \\ \mathit{uu} & \text{otherwise} \end{cases}$$

6.2 Satisfaction and Models

In this subsection we define the satisfaction relation for boolean terms and sentences (closed boolean terms) and also the notion of a model.

Definition 26 Satisfaction.
Let

$\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$	Σ-Algebra
$\nu = \{\nu_k : \chi_k \rightarrow k^{\mathcal{SA}}\}_{k \in K \setminus \{map\}}$	sort environment (order-sorted)
$\eta : \psi \rightarrow \bigcup_{d \in cpo^{\mathcal{SA}}} d$	object environment (unsorted)

and Γ a variable context with $\eta \models_\nu \Gamma$ then:

$\mathcal{A}$ satisfies $(\chi, \Gamma, e, \mathsf{Bool}) \in \mathbf{Form}(\Sigma, \chi, \Gamma)$ wrt. sort environment ν and object environment η (in symbols $\mathcal{A} \models_{\nu,\eta} (\chi, \Gamma, e, \mathsf{Bool})$) iff

$$\mathcal{A} \models_{\nu,\eta} (\chi, \Gamma, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A}[\Gamma \rhd_\chi e :: \mathsf{Bool}]_{\nu,\eta} = \mathit{tt}$$

A special case of the above definition is the satisfaction of sentences. Let $(\chi, \emptyset, e, \mathsf{Bool}) \in \mathbf{Sen}(\Sigma, \chi)$ and η_0 an arbitrary environment, then:

$$\mathcal{A} \models_\nu (\chi, \emptyset, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A}[\emptyset \rhd_\chi e :: \mathsf{Bool}]_{\nu,\eta_0} = \mathit{tt}$$

$$\mathcal{A} \models (\chi, \emptyset, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A} \models_\nu (\chi, \emptyset, e, \mathsf{Bool}) \text{ for every } \nu$$

Now we are able to define models $\mathcal{A}$ of specifications $S = (\Sigma, E)$.

Definition 27 Models.
Let $S = (\Sigma, E)$ be a specification. A polymorphic Σ-algebra $\mathcal{A}$ is a model of S (in symbols $\mathcal{A} \models S$) iff

$$\mathcal{A} \models S \Leftrightarrow \forall p \in E.\, \mathcal{A} \models p$$

7 Conclusions

We have presented the semantics of the kernel part of the Spectrum language. Our work differs in many respects from other approaches. In contrast to LCF we allow the use of type classes. Moreover arbitrary non–continuous functions can be used for specification purposes. This also permits to handle predicates and boolean functions in a uniform manner. In contrast with other semantics for polymorphic lambda calculus (e.g. [15]) we did not provide an explicit type binding operator on the object level. This is not a restriction for languages having an ML–like polymorphism but allows a more simple treatment of the sort language. More precisely we used order sorted algebras instead of the more complex applicative structures. Order sorted algebras were also essential in the description of type classes.

8 Acknowledgment

For comments on draft versions and stimulating discussions we like to thank M. Löwe, B. Möller, F. Nickl, B. Reus, D. Sannella, T. Streicher, A. Tarlecki, M. Wirsing and U. Wolter.
Special thanks go also to our colleagues H. Hussmann, C. Facchi, R. Hettler and D. Nazareth to Tobias Nipkow whose work inspired our treatment of type classes and to Manfred Broy whose role was decisive in the design of the Spectrum language.

References

1. M. Broy. Requirement and Design Specification for Distributed Systems. *LNCS*, 335:33–62, 1988.
2. M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language Spectrum. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.
3. M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language Spectrum. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.
4. R. Milner C Wadsworth M. Gordon. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
5. J. Camilleri. The HOL System Description, Version 1 for HOL 88.1.10. Technical report, Cambridge Research Center, 1989.
6. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.
7. M.-C. Gaudel. Towards Structured Algebraic Specifications. *ESPRIT '85', Status Report of Continuing Work (North-Holland)*, pages 493–510, 1986.
8. M. Gogolla. Partially Ordered Sorts in Algebraic Specifications. In B. Courcelle, editor, *Proc. 9th CAAP 1984, Bordeaux.* Cambridge University Press, 1976.

9. J.A. Goguen and J. Meseguer. Order–Sorted Algebra Solves the Constructor–Selector, Multiple Representation and Coercion Problems. In *Logic in Computer Science*, IEEE, 1987.
10. R. Grosu and F. Regensburger. The Logical Framework of SPECTRUM. Technical Report TUM-I9402, Institut für Informatik, Technische-Universität München, 1994.
11. C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.
12. J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical report, Digital, Systems Research Center, Paolo Alto, California, 1985.
13. P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2).* ACM SIGPLAN Notices, May 1992.
14. J. C. Mitchell. *Introduction to Programming Language Theory.* MIT Press, 1993.
15. J.C. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publisher, 1990.
16. T. Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. CUP, 1993.
17. Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418, 1993.
18. F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL durch LCF.* PhD thesis, Technische Universität München, 1994. to appear.
19. D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, September 1983.
20. G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-Sorted Equational Computation. In *Resolution of Equations in Algebraic Structures.* Academic Press, 1989.
21. C. Strachey. Fundamental Concepts in Programming Languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.
22. P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.